Lecture 3
Data Structures (DAT037)

Ramona Enache
(with slides from Nick Smallbone)

Abstract Datatype (ADT)

ADT = mathematical model of a data type with a certain behaviour

Stacks

A **stack** – stores a sequence of values

Main operations

- `push(x)` – add value x to the stack
- `pop()` – remove the most-recently-pushed value from the stack

LIFO: last in first out

-value removed by pop is always the one that was pushed most recently

Stacks

- Analogy for LIFO: stack of plates
- Can only add or remove plates at the top! You always take off the most recent plate

Stacks

- Stacks are used everywhere!
- Example: **call stack** – whenever you call a function or method, the computer has to remember where to continue after the function returns – it does this by pushing where it had got to onto the call stack

Implementing stacks in Haskell

```
type Stack a = ???
push :: a → Stack a → Stack a
pop :: Stack a → Stack a
empty :: Stack a → Bool

[better API:
pop :: Stack a → Maybe (a, Stack a)]
```

Implementing stacks in Haskell

```
type Stack a = [a]

push :: a → Stack a → Stack a
push x xs = x:xs

pop :: Stack a → Stack a
pop (x:xs) = xs

empty :: Stack a → Bool
empty [] = True
empty (x:xs) = False
```

Implementing stacks in Haskell

In Haskell we don't need to implement a special data type for stacks, as we can use lists instead!

Implementing stacks in Java

- **Idea**: use a dynamic array!
  - push: add a new element to the end of the array
  - pop: decrease size by 1
  - empty?: is size 0?

- **Complexity**: all operations have amortised O(1) complexity
  - Means: although one push may take O(n) time (if the array needs to be copied), this happens rarely enough not to affect the complexity of the whole sequence of operations
  - Formally means: n operations take O(n) time

Queues

- A **queue** is similar to a stack:
  - enqueue(x) – add value x to the queue
  - dequeue() – remove earliest-added value
- Difference: FIFO (first in first out)!
- Value dequeued is always the oldest one that's still in the queue
- Used all over the place – not quite as often as stacks
- Example: controlling access to shared resources in an operating system, e.g. a printer queue

Queues

- Analogy for FIFO: a queue!
- The first person to enter the queue is the first person to leave

Implementing queues in Java

- One idea: a **dynamic array** as before
  - enqueue(x): add x to the end of the dynamic array
  - dequeue(): return first element of array...
    ...but how to remove it?

Question

- What is the time complexity of dequeue implemented in this way ?

a) O(1)

b) O(log N)

c) O(N)

govote.at
Code 285116

Implementing queues in Java – Take 2

- Implement a queue as an **array**, but keep two indices into the array:
  - -**rear**: the index where we enqueue elements
  - -**front**: the index where we dequeue elements
- Compare with stacks, where we had an array plus one index (the top of the stack)
- To enqueue an element, increment rear and put the new element there
- To dequeue, take the element from front and increment front

Question

- What is the problem with this implementation ?

a) repeated insertions/deletions

b) complexity of enqueue

c) complexity of dequeue

govote.at
Code 504731

Implementing queues in Java – Extra

Queues as **circular arrays**

Problem: when rear reaches the end of the array, we can't enqueue anything else

Idea: circular array

- when rear reaches the end of the array, put the next element at index 0 – and set rear to 0
- next after that goes at index 1 front wraps around in the same way

Implementing queues in Haskell

```
type Queue a = ???

enqueue :: a → Queue a → Queue a
dequeue :: Queue a → (a, Queue a)
empty :: Queue a → Bool



[better API:
dequeue :: Queue a → Maybe (a, Queue a)]
```

Implementing queues in Haskell

```
type Queue a = [a]

enqueue :: a → Queue a → Queue a
enqueue x xs = xs++[x]

dequeue :: Queue a → (a, Queue a)
dequeue (x:xs) = (x, xs)

empty :: Queue a → Bool
empty [] = True
empty (x:xs) = False
```

Implementing queues in Haskell – Take 2

```
data Queue a = Queue { front :: [a], rear :: [a] }
   deriving Show

empty :: Queue a
empty = Queue [] []

enqueue :: a -> Queue a -> Queue a
enqueue x (Queue xs ys) = Queue xs (x:ys)

dequeue :: Queue a -> (a, Queue a)
dequeue (Queue [] []) = error "empty queue"
dequeue (Queue (x:xs) ys) = (x, Queue xs ys)
dequeue (Queue [] ys) = dequeue q
  where
    q = Queue (reverse ys) []
```

Stacks and Queues in practice

- Your favourite programming language should have a library module for stacks and queues
- Java: use java.util.Deque<E> – provides addFirst/Last, removeFirst/ Last methods
- For using as a queue, provides add = addFirst, remove = removeLast
- For using as a stack, provides push = addFirst, pop = removeFirst
- Note: Java also provides a Stack class, but this is deprecated – don't use it
- Haskell: instead of a stack, just use a list
- For queues, use also Data.Sequence – a general- purpose sequence data type

Linked Lists

- Inserting and removing elements in the middle of a dynamic array takes O(n) time

(though inserting at the end takes O(1) time)

(and you can also delete from the middle in O(1) time if you don't care about preserving the order)

- A linked list supports inserting and deleting elements from any position in constant time
- But it takes O(n) time to access a specific position in the list

Linked Lists

- Main operations on Linked Lists:

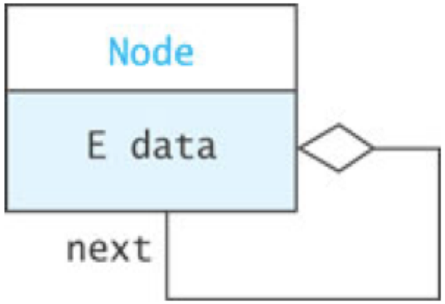
- add a new node (beginning/middle)
- remove a node (beginning/middle)
- iterate
- access a node

Singly-Linked Lists

- A singly-linked list is made up of nodes, where each node contains:
  - some data (the node's value)
  - a link (reference) to the next node in the list

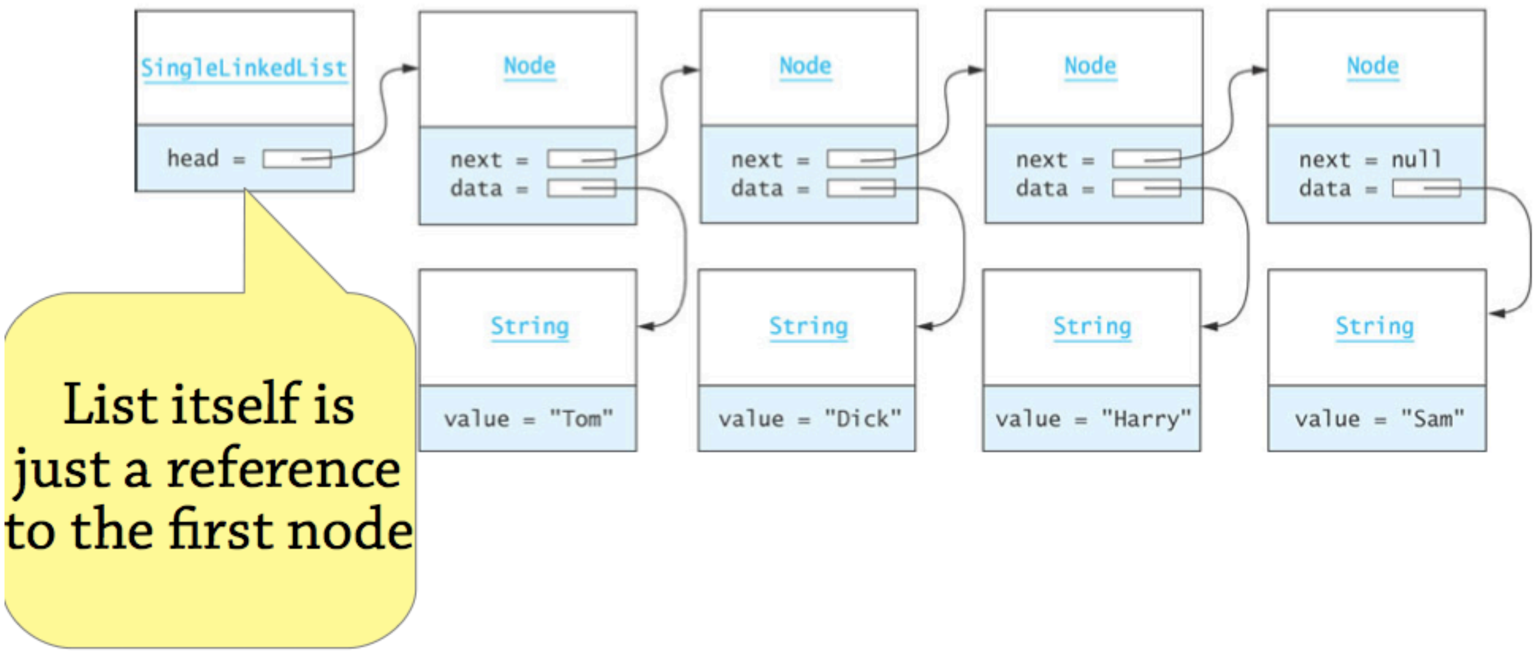
The list also has a special header node !

```
class Node<E>
{ E data;
  Node<E> next; }
```



Singly-Linked Lists - Example

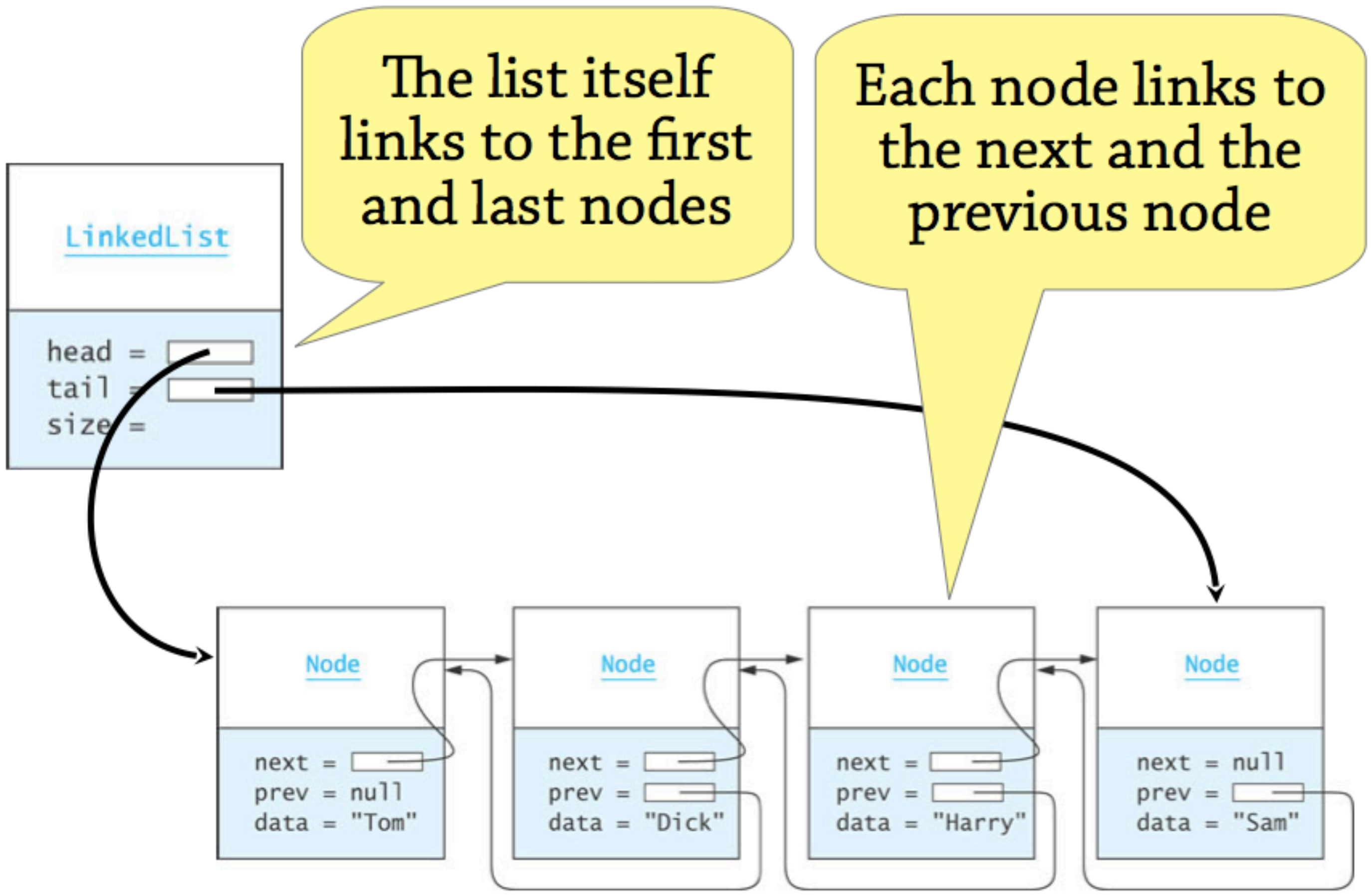
Linked-list representation of the list
[“Tom”, “Dick”, “Harry”, “Sam”]:



Doubly-Linked Lists

- In a **singly-linked list** you can only go forwards through the list:
- If you're at a node, and want to find the previous node, too bad ☹ Only way is to search forward from the beginning of the list
- In a **doubly-linked list**, each node has a link to the next and the previous nodes
- You can in O(1) time:
  - go forwards and backwards through the list
  - insert a node before or after the current one
  - modify or delete the current node
- The “classic” data structure for sequential access

Doubly-Linked Lists - Example



Implementing Linked Lists in Java

- LinkedList<E> class
  - generic doubly-linked lists
  - implements all linked lists + queue operations

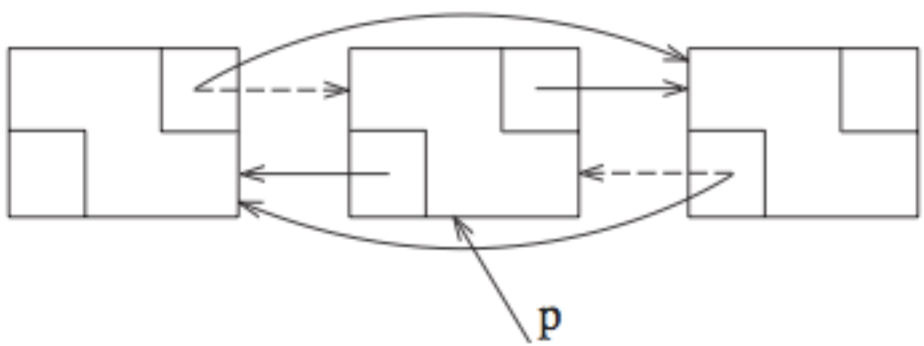
Or

Make your own

Example

- Delete node from double-linked list (3.5)

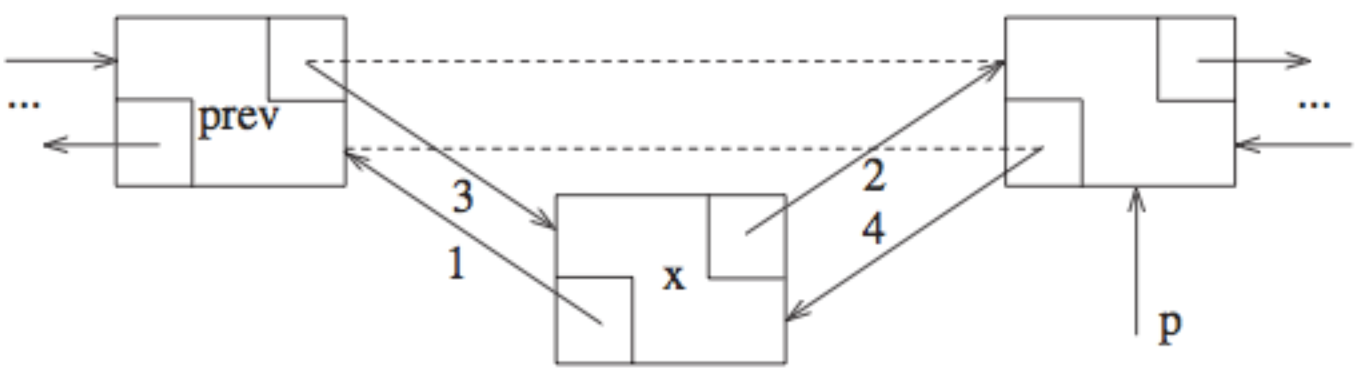
```
p.prev.next = p.next
p.next.prev = p.prev
```



Example

- Insert node in double-linked list (3.5)

```
Node newNode = new Node(x,p.prev,p) //1,2
p.prev.next = newNode; //3
p.prev = newNode; //4
```



Implementing Linked Lists in Haskell

Singly-linked lists

- [a] (normal Haskell list)
- two type of cells – null (end cell) and cons cells
- a list is represented by the pointer to the first cell

Example:

[3,4,5] – Cons 3 (Cons 4 (Cons 5 Null))

1:2:xs – Cons 1 (Cons 2 xs)

Implementing Linked Lists in Haskell

- Ordinary Haskell lists can't be updated (**persistent data structure**)
- When we add an element to a list, the old list stays (until it's garbage collected)
- **Pros**
  - we don't need to copy lists
  - parallel programming can be implemented more easily
- **Cons**
  - some operations could be less effective
- Persistent data structures could be implemented in Java also!

To Do

Read from the book

+ linked lists, stacks, queues (Chapter 3)

Implement:

+ stacks/queues/linked lists in your favourite programming language

Use ADTs for applications:

+ closing brackets using stacks

+ represent polynomials as linked lists and implement basic operations

+ implement a queue with two stacks

Send your lab before November 12th, 23:59

Next time – 14/11

Guest lecturer – Nils Anders Danielsson

Topic - Priority queues (heaps), binary heaps, leftist heaps